09 | 性能模式（上）：如何有效提升性能指标？

2021-06-05 尉刚强

《性能优化高手课》 课程介绍 >

讲述：尉刚强
时长 20:55 大小 19.17M

你好，我是尉刚强。

构建高性能的软件，可以说是所有程序员的共同追求。不过，当我们碰到性能问题时，一般都只会想到数据结构和算法，而忘记系统性能是由运行态的各个硬件所承载的。比如说，当你的代码中遇到一个复杂数学计算的时候，你可能只会想到通过修改优化算法的方式来提升性能。可实际上，一个潜在更优的解决方案是提前计算好放到内存中，使用的时候直接取用，这时候具体的算法性能已经不是最重要的了。

所以，当我们从**硬件运行态的视角**去思考问题的时候，其实就会很容易地找到一些之前发现，但性能收益比较大的解决方法，这就是我接下来的两节课要给你介绍的性能模式。

所谓的性能模式，就是**在软件设计的过程中针对一些特定的上下文场景，以性能提升作为出发点的通用解决方案。**下面我所讲解的各种性能模式，在很多场景下都已经被实际验证过了，你可以在特定的性能优化场景下去套用和实现，从而少走一些性能优化上的弯路。

另外，与软件设计模式不同，一般来说这些性能模式都是比较独立的，你可以将其看作是在时间、空间等不同维度去解决问题的参考思路，在很多的场景下都可以同时使用。所以我希望，你也不要局限于这几种性能模式，而是可以**借鉴这些性能模式的解题思路，来掌握这种全局性地、软硬协同地、动态地思考和解决问题的思路。**

好了，接下来，我就根据我的实践经验，按照常用度从高到低的顺序，来分别给你介绍下快速通道模式、并行分解模式、批处理模式、弹性时间模式、预计算模式、耦合模式、搬移计算模式、丢弃模式这八种性能模式的优化实现原理。这节课我们先来了解下前四种性能模式。

另外，在开始介绍之前我还要说明一点，通常情况下性能模式的抽象级别会比较高，不太适合直接使用举例。为了更清楚地给你展示性能模式的核心思想，我在课程里单独增加了代码示例，这些代码示例很多时候并没有实际意义，你千万不要抄作业且不改名字。

快速通道模式

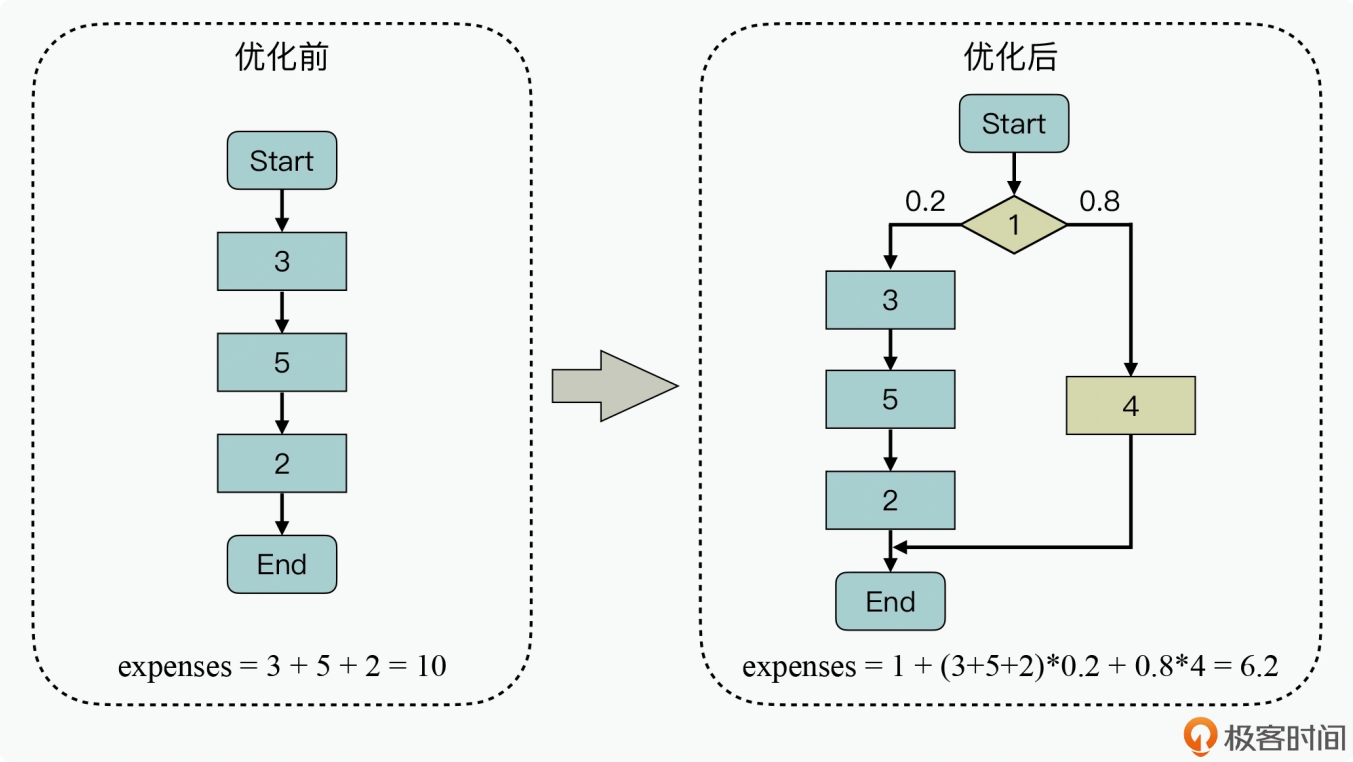
好，我们先来学习下第一种性能模式：快速通道模式。

不知道你平常在逛超市的时候有没有发现这种现象：人们经常购买的商品可能只是整个超市商品的一小部分，大部分商品实际上很少有人购买。

基于这个现象，二十世纪初意大利的经济学家维弗雷提出了著名的**二八效应**，即 80/20 法则（The 80/20 Rule）：在通常情况下，事物的发展都是由少数决定多数的，体现了不公平的特点。

这个法则在计算机领域中也非常适用：系统中大部分用户使用的通常只是少数的一些业务场景；系统中的大部分性能负载是由少量的代码决定的。二八效应在软件系统中的各个维度不断重复着，所以我们能找到那 20% 的决定性场景，寻找定制化方案，就能在很大程度上提升系统的性能。

而这正是快速通道模式的核心思想，下面我们就来看看它的具体实现流程：



图上矩形块中的数字代表的是执行开销，我们可以看到在优化前，系统的总执行开销为 10。而通过分析业务流程和度量数据，我们会发现绝大部分用户使用的少量典型场景，其实可以找到简化处理的方案。

因此，如图上的右半部分所示，针对少数典型场景定制化实现的方案，其执行开销从原来的 3+5+2 变成了 4，系统的总执行开销变为了 6.2，可以说性能提升非常明显。

不过，从解决方案上我们也可以看出，系统在优化过程中增加了额外的业务流程，所以它的业务复杂度就提升了。这也是性能优化实现中经常需要面对的问题，为了追求极致的性能，而不得已舍弃了软件实现的部分简洁性。

这里我给你举个例子。下面是一个数学阶乘的递归实现，我们能看出程序在运行过程中会触发多次函数的调用，这就导致执行时间会比较长（注意，该实现在 num 数字较大的时候会计算越界，不过这不是我们的关注点，可以先忽略）。

复制代码

```
long factBeforeOptimize(int num)
{
    if ( num == 1)
    {
        return 1;
    }
    else
```

```
    {
        return num * fact(num - 1);
    }
}
```

在对这个函数使用统计数据分析后，可以发现 80% 以上的接口调用传入的参数都是 8，所以我们就可以针对这个典型场景实现快速处理，以此来提升性能。修改后的代码实现如下：

复制代码

```
long factAfterOptimize(int num)
{
    if (num == 8)
    {
        return 40320;
    }
    else if (ip == 1)
    {
        return 1;
    }
    else
    {
        return ip * fact(ip - 1);
    }
}
```

通过对比前面的代码，我们能发现改写后的代码实现只是增加了很小一段代码，但系统处理性能却有了很大程度的提升。所以，**快速通道模式的核心思想，就是找到系统中频繁使用的典型场景，然后针对性地提供定制化方案来优化性能。**

不过实际上，针对 CPU 资源紧张、内存资源相对充足的场景，前面所讲的阶乘计算实现是非常糟糕的。因为在该场景下，首先递归运算本身是非常低效的；其次，如果阶乘运算输入参数比较少的话，我们其实可以通过查表来减少执行的运算量。

因此在实际的场景中，快速通道模式的典型应用主要有两类：

一类是数据库 Cache 场景。业务代码经常访问的数据往往是全量数据中很少的一部分，所以在内存中对这部分数据进行 Cache，就可以获得比较明显的性能改善。

另一类针对的是 Web 业务用户经常访问的页面。在前端交互设计中，你可以通过给用户提供快速入口来直接跳转至常用页面，减少很多中间过程页面的访问，从而能够减少系统整体的业务负载。

但是，快速通道模式也存在一定的**局限性**，也就是如果对典型场景分支的预测有错误，就可能会导致系统性能更加恶化。比如，有些数据库 Cache 场景由于 Cache 的命中失败率太高，如果继续引入 Cache，反而会恶化性能表现。

看到这里，你可能还存在一个疑问，那就是**快速通道模式只能用于优化时延吗？**

显然不是的！我一直强调，我们不应该只看到性能模式的外部表现形式，而是应该理解性能模式背后的思维逻辑。

我曾经设计过一款内容资源受限的系统应用，该系统中包含了很多用户实例，而且每个用户的实例结构体比较大，内存总开销超过了设备的内存上限。而通过分析发现，大部分的用户实例其实只使用了结构体内的少部分字段，因此针对用户实例，我设计了精简版结构体和复杂版结构体，由于大部分用户实例使用了精简版结构体，所以系统总内存开销就被控制在了规定的范围内。

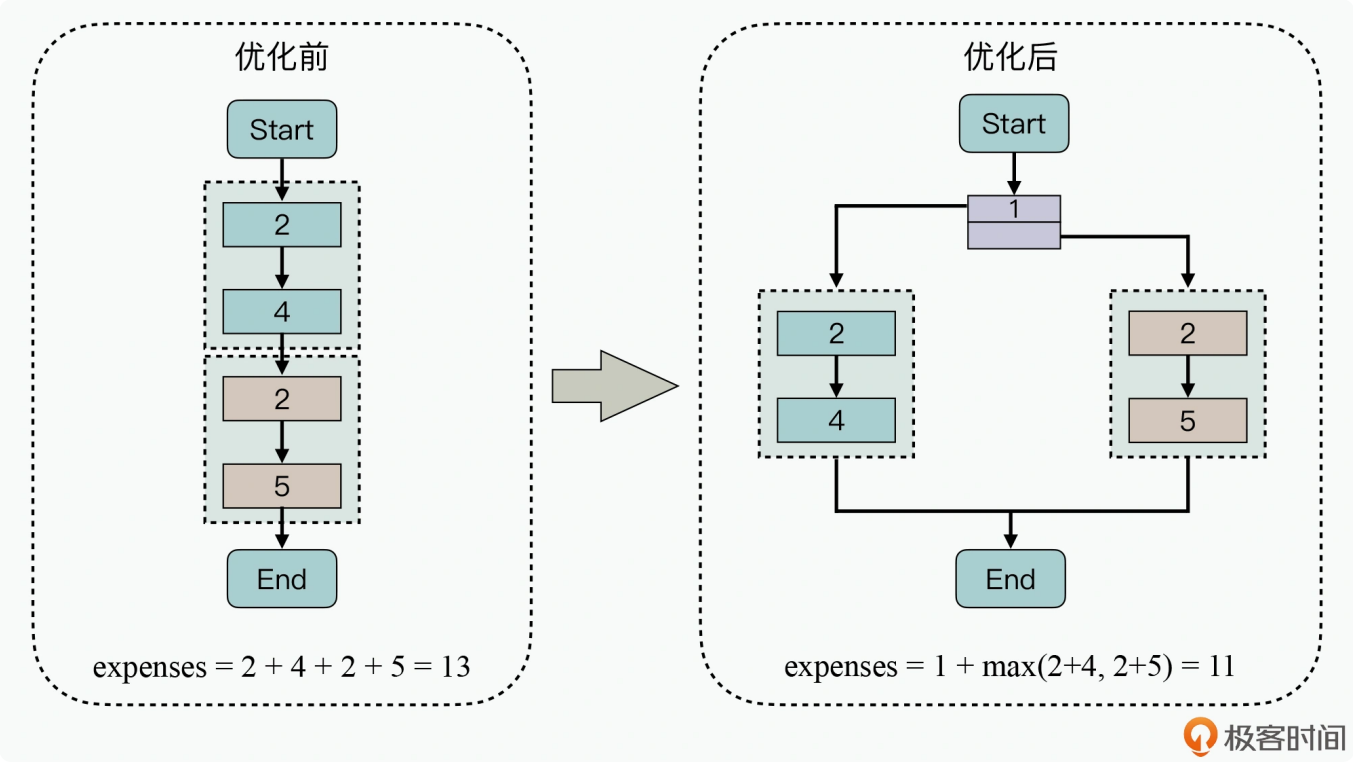
所以你看，这就说明了快速通道模式其实也可以用于内存的优化设计。

并行分解模式

OK，我们接着来看第二种性能模式：并行分解模式。并行其实是一种常态化的解决问题的模式，比如在工作当中，一件任务会被拆分成很多份，交付给不同的人，通过并行来加速任务的完成。

随着计算机技术的不断发展，并行计算也越来越方便可得，因此我们可以在业务处理的过程中，去挖掘并行执行的部分，借助并行计算来优化性能。

这种并行分解的性能解决方案的主要目标，是**减少业务的处理时延**，虽然在调整优化后，系统的执行开销可能会增加，但是从用户的角度来看，处理时延却降低了。下面我们就具体来看看：



如上图所示，矩形方块中的数字表示执行开销，在正常业务流程中可以分为两个大的代码块。在原来的业务流程中，虽然业务是串行执行的，但通过分析业务流程和度量数据后，我们会发现这两个代码块之间是独立可并行的。

所以，图上左侧优化前的总执行开销为 13，借助并行分解模式，将两个代码块并行执行，优化后的总开销降为 11。

不过从图上我们也能看出，右侧优化后的应用占用的 CPU 资源为 1+2+4+2+5=14，总执行消耗资源变多了。但从用户的角度来看，其实是可以感知到处理时延变低了。

下面我们来看一个具体的例子，这是一个使用并发性能模式前后的实现代码对比样例（为了简化代码内容，我隐藏了 loadStudents() 和 loadTeachers() 的具体代码）：

复制代码

```
void loadStudentsAndTeachersBeforeOptimize()
{
    loadStudents(); //加载学生信息列表
    loadTeachers(); //加载老师信息列表
}

void loadStudentsAndTeachersAfterOptimize()
{
    std::thread first(loadStudents); //创建独立线程加载学生信息列表
    std::thread second(loadTeachers); //创建独立线程加载老师信息列表
```

```
11      first.join();
12      second.join();
13  }
```

如上述代码所示，loadStudentsAndTeachersBeforeOptimize 中，两个代码块 loadStudents 和 loadTeachers 之间是可以并行执行的。loadStudentsAndTeachersAfterOptimize 中使用了两个独立线程，分别加载学生和老师信息，从而加速了系统的处理时延。

注意，这里我使用线程来举例，只是为了给你说明并行模式的核心价值，并不是只有线程可以提升并发。要知道，**并行任务分解模式通常是站在系统运行的视角来审视系统，寻找并发增益的。**在第 2 讲中我已经给你介绍过并行系统设计的相关话题，你可以去回顾复习下。

我给你举两个典型的场景案例吧：

> 很多大型数据库的计算引擎就是典型的应用案例。在大数据处理场景中，根据数据拆分、并行计算再规约的典型框架，本质上就是通过增加计算资源实现处理时延降低的目标。
>
> 基于控制流的并发处理也很有价值。这里我以 Web 后端服务的异步机制为例，很多后端服务支持系统同一时刻触发多个 REST 请求，通过并行异步回调处理，从而就在很大程度上降低了后端的处理时延。

不过，并行分解模式使用会引入一些额外的挑战。一般来说，我们习惯使用串行编程思维来解决性能问题，这时心智模型会比较简单，而修改为并行方式后不仅程序的复杂度会提升，还很容易引起很多并发互斥问题。

而且并不是并发度越高，系统的性能就越好，人月神话就是一个很好的反例。一个项目的实现可能需要 10 人 / 月，但这并不意味着让 300 人来做就可以在一天内完成该项工作。

OK，接着这个思路，我们再来思考一个问题：**在单核场景下，有必要创建线程吗？**

我认为实际上，在单核场景下你可能也需要创建线程。早期的操作系统在创建线程时，主要想解决的是 CPU 资源共享的问题。当时线程工作的特点是：少许 CPU 操作和较多异步

IO 操作，在这种场景下创建多个线程时，能够实现当某个线程阻塞时，CPU 资源可以分配给其他线程，从而提升系统的整体性能。

但随着计算机 CPU 硬件多核技术的普及，在很多场景下，线程的工作目标主要在于充分发挥 CPU 硬件多核的价值，线程的工作特点也向不间断执行 CPU 指令的方向演进，因此并行设计的架构与模式也发生了很大的变化。

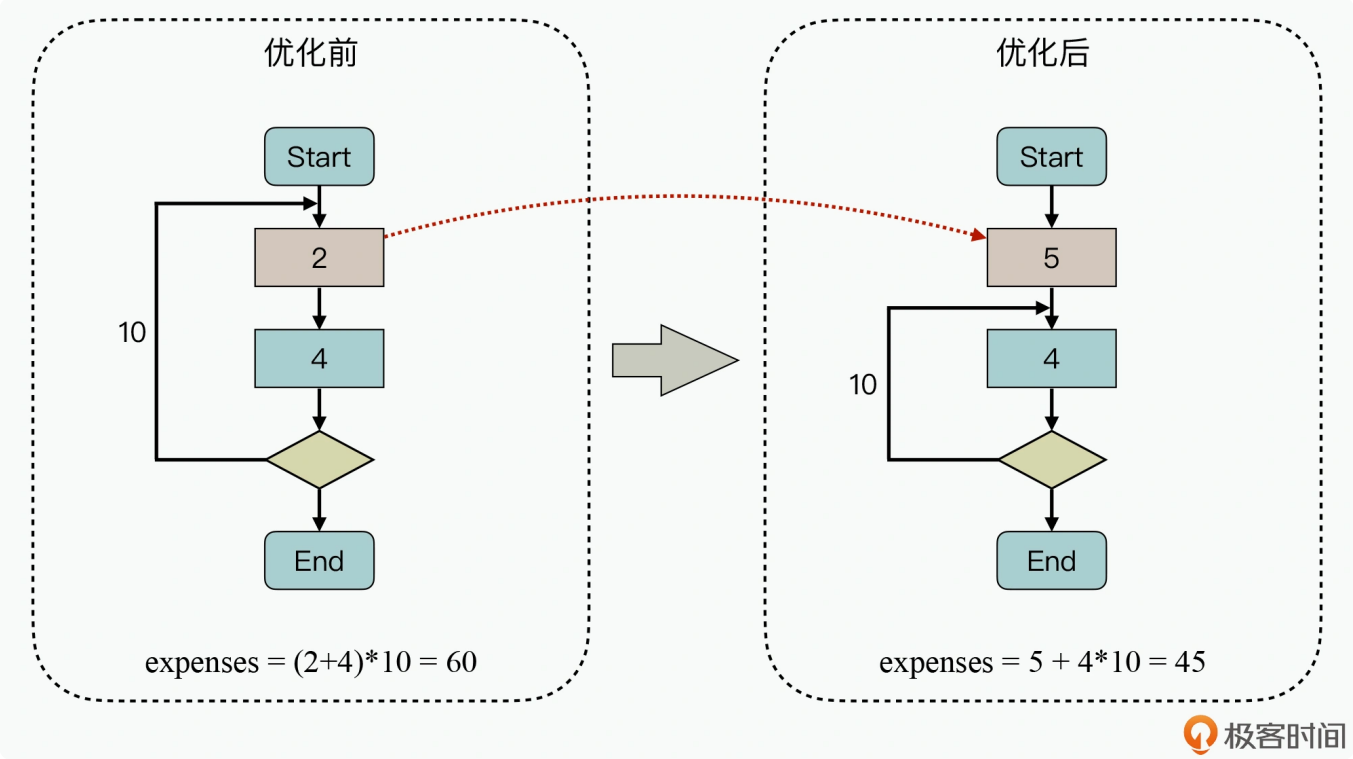
批处理模式

好，我们接着来学习第三种性能模式：批处理模式。

这里我们先来思考一个问题：当我们去超市购买商品时，每次只买一件商品，然后分别购买 10 次所用的时间，跟去超市一次性买足 10 件商品使用的时间，是不是肯定不一样？。

在计算机领域中也是同样的道理，同样的计算任务，分别执行 10 次和一次批处理 10 个，它们需要的时间也是不一样的。所以找出这样的典型业务场景，通过采用批处理方式，就可以很大程度地提升系统性能。

接下来我们就具体看看这种性能模式的优化特点：



如上图所示，左侧优化前循环中第一个矩形代码块执行开销为 2。这个操作在循环体执行了 10 次，需要消耗的计算资源为 20，当修改成批量计算后，执行开销从 20 降低到了 5，从而极大地提升了性能。

另外，从图上我们也能看到，左侧优化前的执行开销为 60，而优化后开销为 45，总的性能提升也比较明显。

同样地，我们还是来看一个具体的例子。这是一个用 C 语言实现的案例，案例中对比了循环遍历处理数据和批处理两种不同的代码实现：

复制代码

```
typedef struct Student
{
    int age;
    char name[20];
} Student;

Student students[10];

void init(Student *student)
{
    student->age = 0;
    memset((void*) student->name, (int) 0, sizeof(student->name));
}

void initAllStudentsBeforeOptimize()
{
    for (int i = 0; i < 10; i++)  // 循环遍历10次初始化操作
    {
        init(&students[i])
    }
}

void initAllStudentAfterOptimize()
{
    memset((void*) students, 0, sizeof(students)) //memset一次性批处理
}
```

可以看到，优化前 initAllStudentsBeforeOptimize 会遍历初始化所有的学生信息，而优化的代码后变成了一次性初始化所有的学生信息。

当然，这个代码只是为了让你更好地理解 1+1 不等于 2 的背后逻辑（请忽略代码中的魔术数字等代码坏味道，这里仅作演示）。同样地，我们也不能只从 CPU 执行时间的视角来看待批处理模式，还可以在内存使用中去整合批量数据的保存，然后通过编码带来很大的内存使用增益。

这里我给你举两个典型的使用场景的例子：

- 针对分析数据库来插入批量数据。对于一些分析性数据库来说，插入一条数据与批量插入上千条数据所使用时间很接近，因此使用批处理模式会带来非常大的优化杠杆。
- 针对网络传输、消息队列传输场景，批量处理模式效果也很明显。发送消息准备时间（如建立连接时间）、结束时间等，都可以通过批量数据发送平均后优化到很小的占比。

当然，批处理模式也存在一些限制。当修改为批量模式之后，会导致部分业务数据的处理延迟增加；同时，批处理模式处理失败所造成的影响通常也会比较大。比如说，你在修改文件的过程中，如果积攒了很多次修改还没有保存，万一突然关机，那么你丢失的信息也就比较多。

最后我们再来思考一个问题：**修改为批处理模式后导致系统的可靠性变差了，怎么办？**

我是这样看待这个问题的，首先，并不是所有的批处理模式都会造成软件复杂度上升，有些甚至会简化软件的实现；其次，当某些批处理模式导致软件复杂度上升时，我们就需要权衡软件复杂度上升所带来性能收益的大小，判断下是否值得这样做。

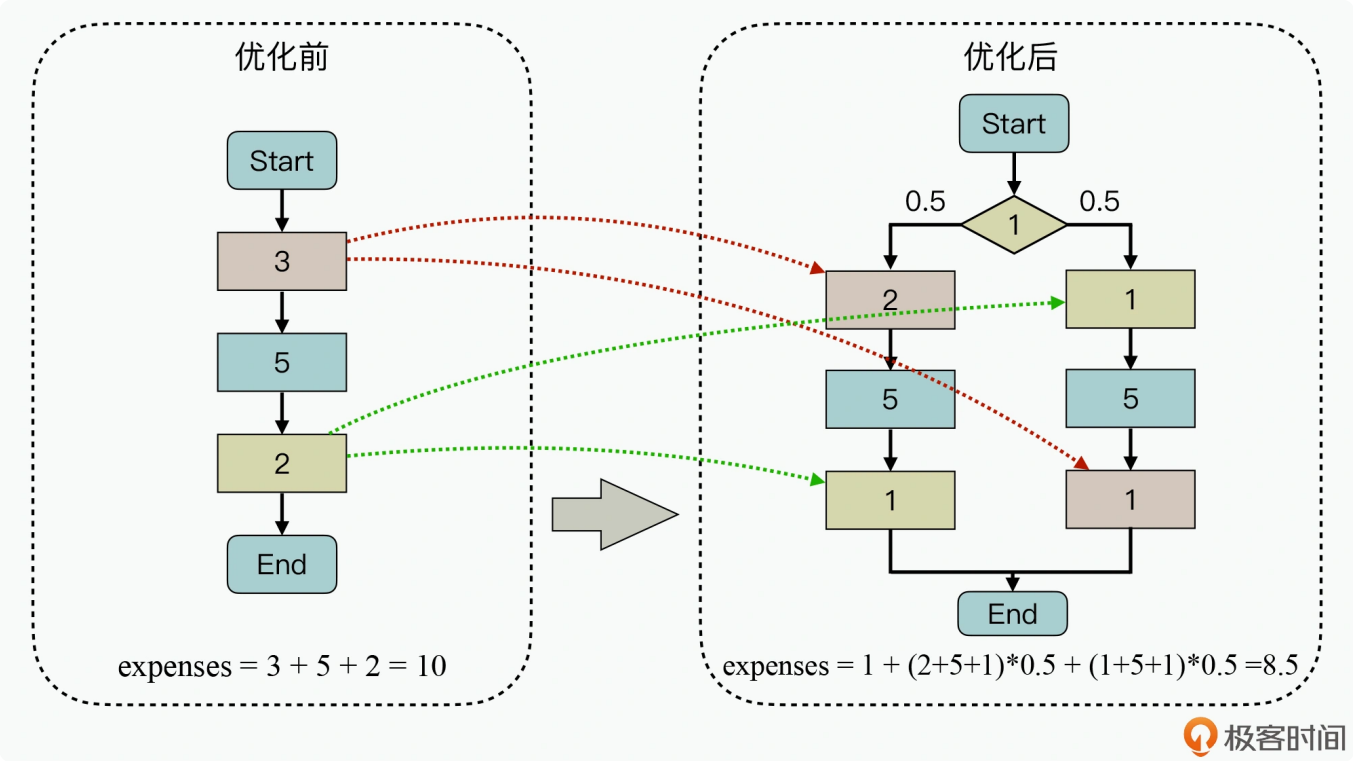
但最后，我还想特别说明一点，并不是复杂的软件可靠性就差，比如大型嵌入式通信系统、银行交易系统等，它们的业务逻辑都非常复杂，但依旧有非常高的可靠性。

弹性时间模式

在介绍这种性能模式之前，我们还是来思考这样一个场景：对职场人士来说，从家到工作单位的距离是不变的，车辆动力也是不变的，但在不同的时间段开到公司的时间却是不一样的。因为上下班时段高峰期，交通拥塞会导致车辆的行驶速度变慢，从而花费的时间就会变长。

回到计算机领域，系统的性能也是由软件和硬件的性能整体决定的，软件服务和硬件也都具有相同的运行特征。在相同的网络带宽下，当链路消息堆积拥塞时，处理时延就会变长。

而弹性时间模式，正是通过离散化业务的请求时间，从而避免系统中的单个软件服务和硬件服务出现拥塞的情况。我们来看下它的具体工作流程：



你会发现，上图中诡异的地方是，优化前第一个矩形代码块的执行开销为 3，只是调整了开始执行时间，优化后的执行开销就降成了 2。

其实这并不神奇，弹性时间模式的原理只是避免了同一时间段上，系统中某个节点发生拥塞而已。

现在我给你举个具体的例子。在下面的代码示例中，我对比展示了确定时间顺序执行和使用弹性时间执行的两种代码实现的差异。

复制代码

```
1 void secondLevelScheduleTaskBeforeOptimize(int second)
2 {
3     syncSendMsgToA();
4     normalwork();
5     syncSendMsgToB();
6 }
```

```
7
8  void secondLevelScheduleTaskAfterOptimize(int second)
9  {
10     if ((second % 2) == 1)
11     {
12         syncSendMsgToB();
13         normalwork();
14         syncSendMsgToA();
15     }
16     else
17     {
18         syncSendMsgToA();
19         normalwork();
20         syncSendMsgToB();
21
22     }
23 }
```

我们知道，系统中同时存在多个进程，且每个进程内都有一个相同的时间点定时处理任务，这种现象是比较普遍的。在函数 syncSendMsgToB() 中，使用消息队列或者网络是被很多进程所共享的，如果在同一时间段内，多个进程同步发送消息，就会因为堵塞导致处理时延增大。

而这里我们可以发现，优化后的代码只是随机调整了 syncSendMsgToB() 和 syncSendMsgToA() 的位置，就可以很大程度上缓解消息拥塞造成的额外时延开销。

一个典型的使用场景，就是系统中针对网络带宽、数据库等平台基础服务，在访问各种定时任务时，通常需要在时间上离散分布来避免拥塞。

而弹性时间模式存在的局限性就在于，在时间、空间上离散分布来提升性能，都需要建立在数据度量的基础上。而且在时间、空间上的离散分布也有可能是不稳定的，随着新业务功能的引入，就可能会导致原来的离散分布调整失效，从而需要进一步地调整。

看到这里，你可能会想到一个问题：**使用任何软硬件资源都要考虑弹性时间分布吗，太麻烦了吧？**

其实也不是，一般来说，当这种资源的使用直接影响到了业务的核心逻辑，并对性能影响较大的时候，我们才需要考虑使用弹性时间的处理方式。

以前我参与设计实现的某通信协议媒体面中，需要与周边三个子系统进行大量信息传递，要共享消息总线。该系统对实时性要求非常高，必须要求与不同子系统在总线上消息交互在时间上错开，才能满足实时性，因此需要定义很严格的消息总线使用时序。但该系统通过网络端口发送的消息，对实时性要求不高，所以就不需要进行弹性时间分布。

小结

性能模式名字并没有像软件设计模式，成为开发人员的通用沟通语言，所以你要重点关注性能模式背后所代表的解决思路与原理，而不是名字。另外，性能模式的抽象粒度可大可小，你可以在不同抽象层级去使用这些模式，但我更推荐在更大的抽象级别去使用性能模式。

性能模式是建立在系统运行视图的基础上，因此你需要对软件执行模型有深入的理解，同时还要借助数据度量来辅助分析，否则很容易引起适得其反的效果。

思考题

面向对象设计模式基于 SOLID 原则选择，SOLID 原则基于正交设计，而正交设计的底层原理又是高内聚、低耦合的，那么你认为性能模式底层的基础逻辑是什么呢？

欢迎给我留言，分享你的思考和看法。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 可监控设计：如何利用eBPF来指导可监控设计？

下一篇 10 | 性能模式（下）：如何解决核心的性能问题？

精选留言 (4)

写留言

型火
2021-06-07

底层逻辑我认为是资源与收益的权衡。更像经济学，用更少的成本带来最大的收益，比如多开并行的worker可以提高效率，但又会带来新的复杂度，需要权衡，折衷，trade-off. 换句话说，降本增效。

作者回复: 哈哈，感觉每次你总结的更加精炼！赞

1

型火
2021-06-07

作者的抽象能力太牛了，这个专栏就是宝藏。这种高屋建瓴，合纵连横的技术抽象很见功力，广度和深度。更犀利的是，不只是纸上谈兵，还有细化的画图分析以及每种模式的优点缺点，解决了什么问题，又带来了什么问题。有全局，有细节，有trade-off,有实战。仰望星空，脚踏实地。最可贵的是有些加粗文字的凝练和准确，可见是经过深思熟虑和创造的。哈哈，不是拖～

展开

作者回复: 很开心，收到你的肯定，我会继续努力分享自己的心得和经验，希望对你有帮助。

1 1

该溜子
2021-07-11

有没有更具体点的case呢

展开

danger boy
2021-06-11

1、快速通道模式
频繁使用的场景，针对性的提供定制化方案优化性能。
缓存的利用，空间换时间，把数据提前处理好，用的时候直接去缓存拿！

2、并行分解模式...

展开

作者回复: 总结很好！赞